Machine Learning - Enron Dataset for Fraud Detection

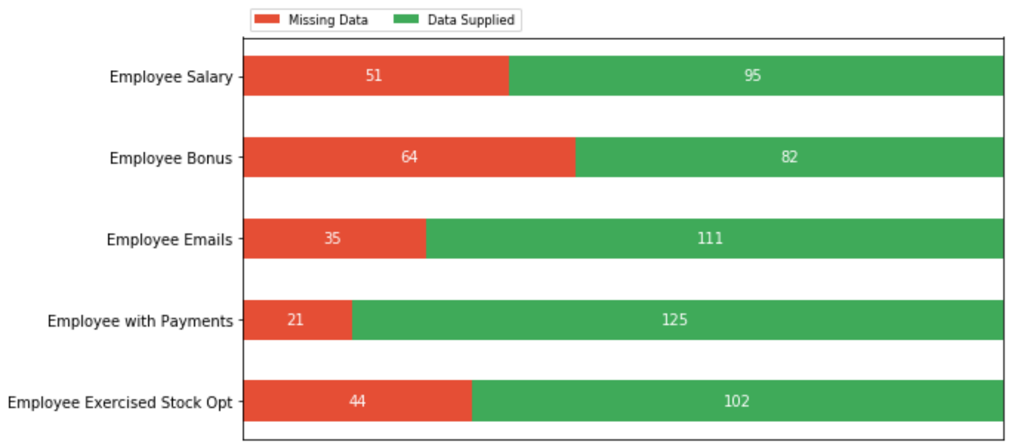
Renee Cothern

TOC:

Summarization and Data Cleaning

The overall goal is to build a "person of interest" identifier based on the public Enron financial and email dataset which was made public as a result of the scandal. Machine learning can be useful for building an identifier of this type because it can identify patterns and trends in the data that might not necessarily be apparent just by observation, and use it to predict persons of interest that might be involved in fraud. Since the Enron dataset already provided training labels of POI (person of interests) vs. non-POI's to classify the data with, a supervised training methodology is what was needed.

In regards to the data, total number of employees were 146, with 21 categories (features), and 18 people of interest (poi). There were a few **outliers** I noticed. There were two within the "employee names" - one was named "TOTAL", probably as the result of adding together the results, and another named "THE TRAVEL AGENCY IN THE PARK" which I removed. I also chose to replace NaNs and negative numbers for all the financial data that I was using in my features list with zeros. Originally, there were 51 employees with a salary of **NaN**, 64 employees with a bonus of **NaN**, and 21 **NaN** payments.



NOTE: Figure done with ***pyplot*** and included in code

Feature Selection for Optimal Machine Learning

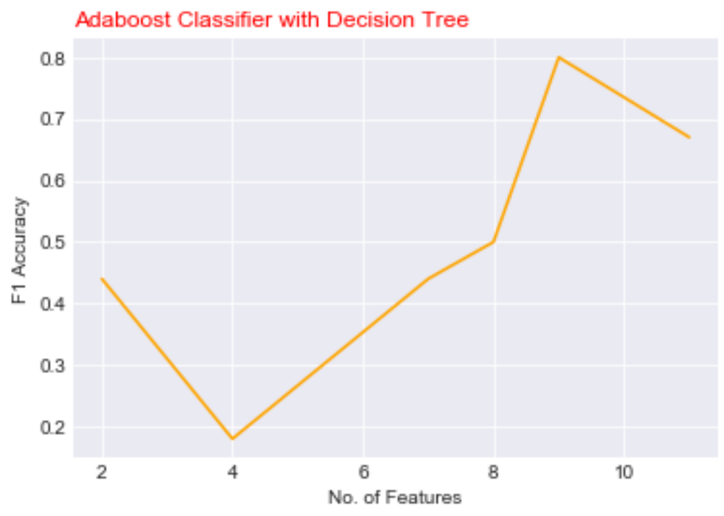
The features I chose to use were initially ones that I intuitively thought were of core importance to the dataset, poi, salary, bonus and total payments.

The other features I chose using SelectKBest scoring (algorithm that selects features that are most powerful and returns a weighted value) selecting the 10 best feature scores, and I put this in a function called: **best_features_suggestion**. The result of this function was as follows:

Feature Test	Feature Score
Exercised Stock Options	24.250
Bonus	20.257
Salary	17.718
Fraction to POI	15.946
Total Payments	8.571
Shared Receipt with POI	8.276
Loan Advances	7.067
From POI to this Person	5.041
Fraction from POI	2.964
From This Person to POI	2.295

NOTE: Highlighted items were eliminated from final features.

In addition, I also tested the F1 accuracy score for the classifier of my choice. I used a balance of the feature score and the F1 accuracy score to come up with the best number of features. I eliminated features based on the weighted feature score above. With POI feature, that makes 9 features.



NOTE: Figure done with ***pyplot*** and included in code

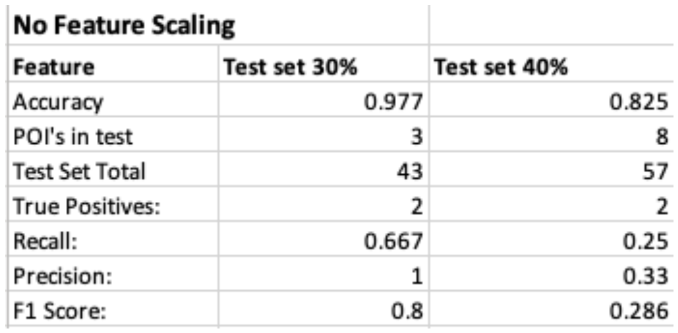
I then **engineered two new features** that I added to my features_list, called:

- fraction_from_poi
- fraction_to_poi

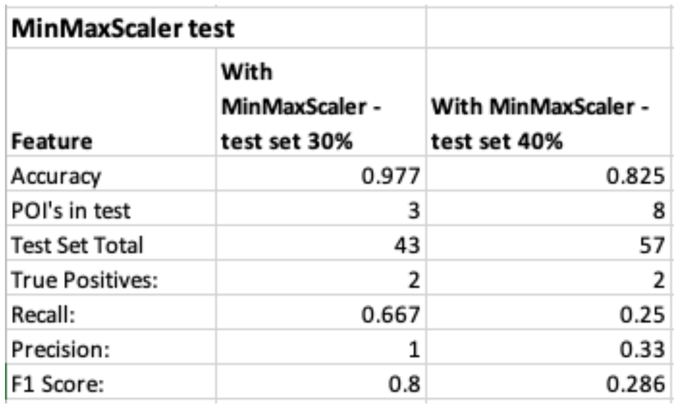
These two features are a proportion created by dividing "from messages" and "to messages" by the total of all messages. Once adding them to my features_list I ran my **best_features_suggestion** function again so I could further refine my list. Upon reviewing the feature scores again, I decided to remove **from_poi_to_this_person**, **from_this_person_to_poi** as they seem extraneous with my new features and rated fairly low when I ran my engineered **best_features_suggestion** tool again.

Testing the Data with Scaling:

I tested the data set using MinMaxScaler to put the data on an even scale from 0 - 1. However, I noticed it yielded absolutely no difference. (see tables below)

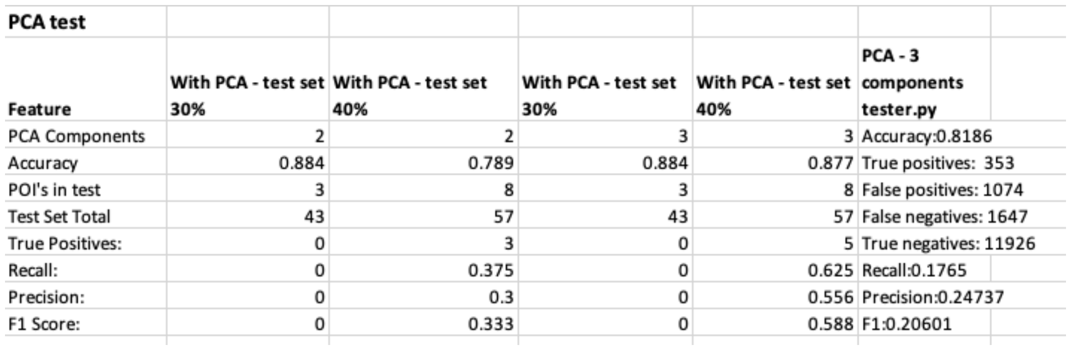


No Feature Scaling		
Feature	**Test set 30%**	**Test set 40%**
Accuracy	0.977	0.825
POI's in test	3	8
Test Set Total	43	57
True Positives:	2	2
Recall:	0.667	0.25
Precision:	1	0.33
F1 Score:	0.8	0.286

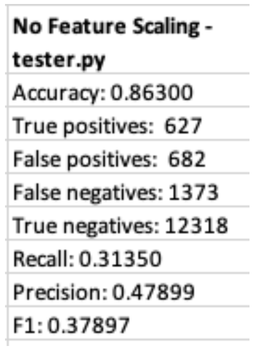


MinMaxScaler test		
Feature	**With MinMaxScaler - test set 30%**	**With MinMaxScaler - test set 40%**
Accuracy	0.977	0.825
POI's in test	3	8
Test Set Total	43	57
True Positives:	2	2
Recall:	0.667	0.25
Precision:	1	0.33
F1 Score:	0.8	0.286

I also tested reducing the data dimensionality using PCA, with both 2 and 3 components. At first it appeared it was making a difference as I enlarged the dataset. However, after using tester.py (testing tool), I noticed the recall and precision scores went down, as well as the number of true positives. (see tables below)



PCA test					
Feature	**With PCA - test set 30%**	**With PCA - test set 40%**	**With PCA - test set 30%**	**With PCA - test set 40%**	**PCA - 3 components tester.py**
PCA Components	2	2	3	3	Accuracy:0.8186
Accuracy	0.884	0.789	0.884	0.877	True positives: 353
POI's in test	3	8	3	8	False positives: 1074
Test Set Total	43	57	43	57	False negatives: 1647
True Positives:	0	3	0	5	True negatives: 11926
Recall:	0	0.375	0	0.625	Recall:0.1765
Precision:	0	0.3	0	0.556	Precision:0.24737
F1 Score:	0	0.333	0	0.588	F1:0.20601



No Feature Scaling - tester.py
Accuracy: 0.86300
True positives: 627
False positives: 682
False negatives: 1373
True negatives: 12318
Recall: 0.31350
Precision: 0.47899
F1: 0.37897

Algorithm Selection and Performance

The algorithm I chose by iterating through a list of algorithms printing out the time to learn and predict, the accuracy and the f1 score of each algorithm, as well as the settings for each algorithm so that I could choose to change or add parameters to fine tune to give the best accuracy scores.

My algorithm list I iterated through were **Naive Bayes, Support Vector Machine, Decision Tree and AdaBoost**.

```
classifiers =
****************************************
Classifier: GaussianNB(priors=None, var_smoothing=1e-09) -->

training time: 0.001 s

prediction time: 0.0 s

accuracy 0.930:

f1 score: 0.5714285714285715

****************************************


****************************************
Classifier: SVC(C=10000.0, cache_size=200, class_weight=None, coef0=0.0,
 decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
 kernel='rbf', max_iter=-1, probability=False, random_state=None,
 shrinking=True, tol=0.001, verbose=False) -->

training time: 0.002 s

prediction time: 0.001 s

accuracy 0.930:

f1 score: 0.0

***************************************


***************************************
Classifier: DecisionTreeClassifier(class_weight=None, criterion='gini',
max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best') -->

training time: 0.002 s

prediction time: 0.001 s

accuracy 0.837:

f1 score: 0.2222222222222222

***************************************


***************************************
Classifier: DecisionTreeClassifier(class_weight=None, criterion='gini',
max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=40,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best') -->

training time: 0.001 s

prediction time: 0.002 s

accuracy 0.860:

f1 score: 0.4

***************************************
```

```
***************************************
Classifier: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=4,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=40,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best') -->

training time: 0.002 s

prediction time: 0.0 s

accuracy 0.860:

f1 score: 0.4

***************************************


***************************************
Classifier: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=50, random_state=None) -->

training time: 0.163 s

prediction time: 0.008 s

accuracy 0.930:

f1 score: 0.4

***************************************


***************************************
Classifier: AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion='gini',
max_depth=4,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=20,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=1.0, n_estimators=60, random_state=0) -->

training time: 0.178 s

prediction time: 0.008 s

accuracy 0.977:
```

```
f1 score: 0.8

*****************************************
```

Ultimately, the last AdaBoostClassifier in this list is what I choose because it had the best accuracy and f1 score. I found it interesting how some of my classifiers had a high accuracy, however the f1 score was low - or vice versa. This means that in some cases the algorithms ranked the data okay or not okay, but then the f1 score says that it didn't do very well on the rest of the test set creating false positives.

Parameter Tuning

When you tune the parameters of an algorithm, you are tuning the settings of the model to discover the most likely predictions. If you don't do this well, the result could be that future data inputs for a machine to learn from won't be properly placed for prediction. This could result in bad decision making.

I chose to add Decision Tree parameters to AdaBoost as well so I had some control over the decision stumps. A weak classifier (decision stump) is prepared on the training data and through several iterations it finds the best classifier through weighted scores of predictability.

Here is **what I chose to use**:

```
AdaBoostClassifier(DecisionTreeClassifier(min_samples_split=20, max_depth=4),
algorithm="SAMME.R", n_estimators=60, random_state=0)
```

Validation

Validation is the process where a trained model is evaluated with a testing data set - creating a separate training vs. testing set. But unfortunately there is a trade-off:
If you increase your test set, you get better validation but less learning result, and vice versa.

In order to validate my analysis, I had to experiment with changing the size of the testing set. When I changed the testing set from .3 to .4 proportionally, I noticed that I did get better accuracy scores. However the preciseness of predicting the test data went down.

Evaluation Metrics

The 2 evaluation metrics I used were recall and precision. **Recall** basically says, out of all the items that are truly positive, how many were actually correctly classified as positive? Or simply put, it "recalls" how an individual concept is doing. **Precision** says, out of all the items labeled positive, how many truly ARE positive and belong to the positive class? In other words, how "precise" was the prediction?

For my test, set, here were the evaluation metrics:

```
POI's in the test set:   3
Number of true positives:  2
Recall: 0.667
Precision: 1.000
f1 score: 0.800
```

So in this case 67% of the POIs in the test set were predicted correctly as a POIs (recall) and there were 2 true positives. In regards to precision, 100% of my predicted test set were classified correctly. F1 score is the average of recall and precision, and I found that the f1 score was a useful tool in finding a good classifier and evaluating my features.

Using **tester.py (testing tool)**, the results were a bit different due to the different size of the testing/training.

```
    Accuracy: 0.86300
    Precision: 0.47899
    Recall: 0.31350
    F1: 0.37897
    F2: 0.33677
    Total predictions: 15000
```